Automatic Parallelization:
A Comparison of CRAY fpp and KAI KAP/CRAY

RND-90-010

Douglas M. Pase[1]
Katherine E. Fletcher
Computer Sciences Corporation
NASA Ames Research Center
Moffett Field, CA 94035

Abstract

In this report we examine two existing commercial parallelizing code restructurers: the CRAY Autotasking[2] *facility and Kuck and Associates' KAP/CRAY. In particular we measure their ability to vectorize and parallelize 25 scientific benchmarks for a CRAY Y-MP supercomputer. We measure the overall code performance, the speedup gained by parallelizing codes with these products, and the overhead used in the parallel execution of each benchmark.*

Introduction

Many different ideas about the abilities of parallelizing tools have been put forward over the years. Some claim that existing programs are inappropriate for parallel execution because the programming model used (the Von Neumann model) is inherently sequential, and such codes should be rewritten using either languages designed to express parallelism, or sequential languages with parallel extensions added. Others claim that the codes themselves are adequate, and that code restructuring tools will be able to provide the needed parallelism without the expense of relearning programming techniques and rewriting applications. Still others think that the problem of parallelizing existing codes is not impossible, but it is too complex for automatic tools. For them, interactive code restructuring tools seem to be the way to go.

In this report we explore the second assertion, that is, that existing "dusty deck" programs contain sufficient parallelism, at least for moderately parallel machines, and that it is reasonable to expect a sophisticated compiler to find it. To examine this question, we compare the performance of two of the most sophisticated commercially available code restructurers: CRAY fpp, and Kuck and Associates KAP/CRAY. Performance is measured on a parallel vector supercomputer—an 8-processor CRAY Research, Inc., Y-MP.

Both tools were designed to be used either as automatic code restructuring tools, or as "batch mode" restructurers. We make no attempt to evaluate either tool as a batch tool, that is, we do not manually improve the code based on what the restructurer was or was not able

[1]. This work was supported by NASA Contract No. NAS2-12961 while the authors were employed by Computer Sciences Corporation under contract to the Numerical Aerodynamic Simulation Division at NASA Ames Research Center.

to do. We put all benchmarks through one or both parallelizers and compile the result without further modification. The only exception to this rule is that we correct errors when the restructured code fails to execute correctly. All vectorization and parallelization (CDIR and CMIC) directives originally in the benchmarks were also removed prior to their use.

Benchmarks were selected from both public sources and sources private to NAS. The benchmarks reflect as fair a representation of production codes as possible. The public codes include the Perfect Benchmark Suite [1], Livermore Loops, and the NAS Kernels.

Hardware Environment

All codes were executed on a CRAY Research, Inc., Y-MP. Benchmarks were executed in dedicated time, that is, no other programs were allowed to use the machine while timings were taking place. In this way no external factors, such as memory bank conflicts with other programs, were allowed to interfere with benchmark performance. In addition, all intermediate I/O was sent to the Solid-State Device (SSD) instead of rotating storage, to reduce the impact of I/O on performance. Important Y-MP hardware characteristics are summarized in Table 1. CPU functional units are described in Table 2, and Y-MP register structure is given in Table 3.
A more complete description of the Y-MP hardware system may be found in [2].

Number of CPUs:	8
Clock Period:	6.0 ns
Instruction Buffer Size:	512 16-bit parcels (4 buffers)
Memory size:	128 million 64-bit words (2 ** 30 bytes)
Memory access time:	17 clock periods (107 ns)
Memory bank cycle time:	5 clock periods (30 ns)
Number of memory banks:	256
Number of memory ports/CPU:	4 (2 read, 1 write, 1 I/O)
Solid-State Device:	256 million words 200 Mwords/sec transfer (4 ports @ 50 Mw/s each)

Table 1 — Y-MP Hardware Features

Address Functional Units	Add
	Multiply
Scalar Functional Units	Integer Add
	Shift
	Logical
	Population/Parity/Lead 0
Vector Functional Units	Integer Add
	Shift
	Logical (2)
	Population/Parity
Vector/Scalar Functional Units	Floating Point Add
	Floating Point Multiply
	Reciprocal Approximation

Table 2 — CPU Functional Units

Address (A)	8 32-bit Registers
Intermediate Address (B)	64 32-bit Registers
Vector (V)	8 Registers of 64 elements each, 64 bits per element
Scalar (S)	8 64-bit Registers
Intermediate Scalar (T)	64 32-bit Registers

Table 3 — Y-MP Register Structure

Software Environment

All benchmarks were written in Fortran for a serial vector supercomputer. Translation from serial code to parallel code was performed by CRAY fpp, or Kuck and Associates KAP/CRAY, or both.
Benchmarks were translated from Fortran code with embedded microtasking directives to Fortran with calls to the CRAY parallel library by the utility fmp. The resulting code was then compiled by

```
Fortran to Microtasked FortranCRAY fpp4.0.1
      KAP/CRAY1.01
Microtasked Ftn to parallel codeCRAY fmp4.0.1
Fortran CompilerCFT774.0.1
Operating SystemUNICOS5.1.10
LibrariesUNICOS5.1.10
```

Table 4 — Software System Versions

the CRAY cft77 Fortran compiler. The version numbers for each of the products are given in Table 4.

The Microtasking parallel environment is supported through special calls to the CRAY parallel library. Microtasking supports a model of parallel programming known as *Single Program, Multiple Data*, or SPMD. The idea is that when a program is executed each process will execute the same program, but on different data. Also, each process executes independently of other processes so they need not take the same amount of time to execute, nor even execute the same sequence of instructions.

At the beginning of the execution of a parallel program, slave processes are created and placed in a "parked" state. While in this state they accrue "semaphore wait time." Other processes can use the CPUs while the slaves are in this state, but the slaves have a high priority when they are unparked and return to the run queue. Once running, the slaves resume accumulating "user time."

Parallel regions in the program are placed within CMIC$ PARALLEL and CMIC$ END PARALLEL directives. The directives may be conditional or unconditional. When a parallel region is entered, the master process unparks the slave processes, which are returned to the run queue. Each process then executes the same segment of code in parallel with the others. If the code segment is a parallel loop, iterations of the loop are divided into blocks which are distributed across the available processes. Iteration distribution is done one iteration per processor. At the end of the parallel region, all slave processes are parked once again. Table 5 summarizes the overhead involved with managing the parallel execution for both master and slave processes in clock periods (CPs), and the rough equivalent in instructions.

MASTER:				
	Get CPUs	50-100	msec	(2-4 million instr.)
	Unpark slaves	175	CPs	(44 instructions)
	Get Iteration	25	CPs	(6 instructions)
	END PARALLEL	30	CPs	(7 instructions)
SLAVE:				
	Unpark Slaves	200	CPs	(50 instructions)
	Get Iteration	25	CPs	(6 instructions)
	END PARALLEL	30	CPs	(7 instructions)
	Repark Slaves	75	CPs	(19 instructions)

Table 5 — Parallel System Overhead

CRAY Fpp And KAP/CRAY

The set of compiler options used to compile a program can strongly affect its performance. In this study we had to trade off aggressive parallelization against reliability. Aggressive parallelization held opportunities for greater speedups, but also carried the risk of not functioning for all of the codes. As a compromise we used the most aggressive options that also worked for a reasonable majority of the codes. For fpp we selected -Wd-e46ijt except where noted otherwise. This selection enables fpp switches 4, 6, i, j, and t, which mean:

4	Asserts that private array initial values are not needed.
6	Enables automatic inlining of routines that are less than 50 source lines in length, and do not call other routines.
i	Enables inner loops with variable iteration counts to be autotasked, if analysis warrants it.
j	Replaces matrix multiplication loops with a library call.
t	Enables aggressive loop exchanges to take place.

We used the default command line switch settings for KAP. The default settings were as follows:

MV=3	Sets the minimum vector length to 3.
MVC=1000	Sets the minimum amount of work in a loop that KAP will execute in vector-concurrent mode to 1000 iterations. Two-version loops are created if the loop bounds are unknown. No two-version loops are generated when mvc=-1 is used.
MC=950	Sets the minimum amount of work in a loop that will be concurrentized. Again, two-version loops are generated when loop bounds are unknown.
P=0	When a program is concurrentized, it is always compiled for an unknown number of processors.
DUST=3	Loop re-rolling is enabled, and certain IF/DO code transformations are performed.
LM=21000	Sets a crude upper limit on the amount of work KAP will perform in trying to optimize a loop.
O=5	KAP vectorizes any loop where it is legal to do so. Loop interchanges may be applied, even totriangular loops, and reductions are recognized. Enhanced dependence analysis is used, and KAP attempts to break dependence cycles. Lifetime analysis of variables is performed. Array expansion is performed.
UR=16	Loops are unrolled by at most 16 iterations.
UR2=40	Same as UR, but the limit is a measure of "work" within the loop rather than the number of iterations.
NOEXPAND	Local subroutine inlining is inhibited. Using x=a allows KAP to inline some subroutines and functions. Exd=-1 restricts inline expansion to routines which do not contain function or subroutine references.

Vectorization and parallelization tools rely heavily on techniques for code analysis, primarily dependence and loop analysis, and on code transformations. Loop analysis classifies the type of loop and the existence of dependence cycles between statements. This information is used in determining which loop transformations will be legal and beneficial.

Dependence analysis determines whether loop iterations can be executed independently, or whether reordering them will cause the program to execute incorrectly. Comprehensive dependence analysis for all possible array subscript expressions is very time consuming (it is an *NP-Complete* problem), and therefore must be approximated. Accurate dependence analysis at times requires information about how the program is used, i.e., runtime information. A third factor which may inhibit an accurate analysis is that a dependence may truly exist within a program, but the algorithm in which it occurs may be insensitive to its violation and thus the dependence could be ignored.

Once the dependence and loop analyses have been done, loop optimizations may take place. Generally the optimizations are different types of loop transformations which enhance vectorization or parallelization. Many of the code transformations performed by fpp and KAP will be described below. A complete description of the fpp code transformation techniques may be found in [3]. Code transformations which KAP uses are described in [4]. A summary of each may be found in Table 6.

Benchmarks

As mentioned before, benchmarks used in this study were selected from NAS sources and the Perfect Benchmarks [1]. NAS private codes include 2-D and 3-D aerospace simulation codes, structural codes, and synthetic benchmarks either developed here or in common use at the NAS facility. NAS private codes include seven Navier-Stokes computational fluid dynamics (CFD) codes, of which four involve chemistry, and one solves a supersonic problem. Of the remaining NAS codes, one solves a structural problem for a high-Mach air frame, and two are synthetic FFT codes. A detailed description of each benchmark is given in Table 7.

Benchmark memory requirements vary from as little as 110 Kword to as large as 54 Mword. The distribution of memory requirements for the various benchmarks is shown in Figure 1. A detailed summary of benchmark characteristics is given in Appendix A.

Code Transformation	fpp	KAP
Reduction recognition	yes	yes
Recurrence recognition	yes	yes
Two version loops	yes	yes
Vectorize IF loops	yes	yes
Partial loop vectorization (fission)	yes	yes
Loop collapse	yes	yes
Loop fusion	yes	no
Loop interchange	yes	yes
Loop peeling	yes	yes
Loop unrolling	yes	yes
Loop rerolling	yes	yes
Scalar expansion	yes	yes
Code inlining	yes	yes

Table 6 — Fpp And KAP Code Transformations

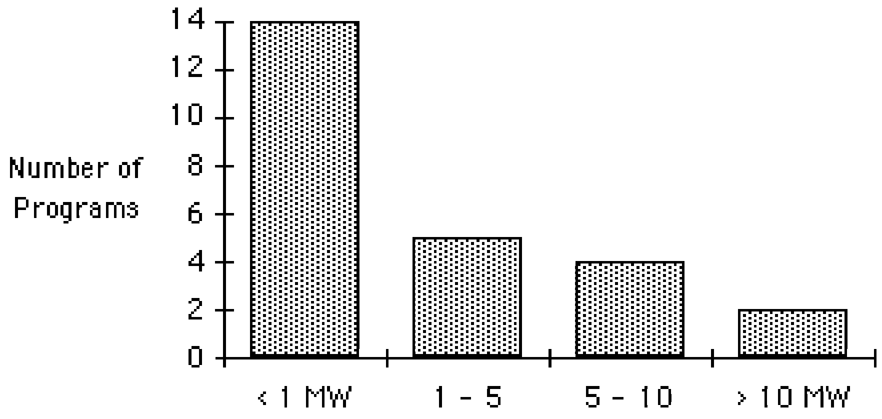


Figure 1 — Distribution of Benchmark Memory Requirements

Total number of floating point operations vary from 58 million to 78 billion operations. Most of the NAS codes have between 10 and 25 billion floating point operations, while the Livermore Loops, NASKERN, and all but one of the Perfect Benchmarks are much smaller than that. The largest code is NAS10, the smallest is SPICE.

Given the available floating point units on the Y-MP, the exact mix of floating point additions, multiplications, and reciprocals is critical to achieving maximum performance. If there are significantly more operations of one type than another, some floating point functional unit will be left with no work while another will have an excess. More precisely, each functional unit should have enough work to fill the same number of clock periods if it is to achieve maximum overlap. One should note, however, that scheduling conflicts and program dependences can further inhibit overlap even when the number of each type of operation is balanced. The next figure shows

1	NAS01	is a general purpose 3-D fluid flow solver for high angle-of-attack problems. It uses an implicit finite difference scheme for solving the viscid, unsteady flow about a sphere. It employs approximate factorization with flux-split differencing in the solution.
2	NAS02	simulates the incompressible laminar fluid flow around a single post mounted between two plates. The Navier-Stokes equations are solved in three dimensions by an approximate factorization algorithm, using pseudocompressibility tosolve the pressure field.
3	NAS03	computes the decimal expansion of the mathematical constant π. The core of this program uses highly vectorizable fast Fourier transforms (FFTs).
4	NAS04	repeatedly computes forward and reverse FFTs on pseudorandom data. The FFTs are highly vectorizable. With hand-inserted vectorization directives, both NAS03 and NAS04 are capable of 150 to 160 million floating point operations per second (MFLOPS) on a single Y-MP CPU.
5	NAS05	gives a time-accurate 3-D simulation of the mixing of reactive fluids. Pseudospectral methods are used to compute the derivatives, and fluid flow incompressibility is modified by adding terms to account for variable density in the fluid.
6	NAS06	gives a time-accurate solution to a 3-D Navier-Stokes problem. The code sweeps entire planes, performing a Gaussian elimination of conserved quantities at each cell.
7	NAS07	uses a fast, accurate, Choleski method for solving 16,000 linear equations. The test case used is the structural analysis of an air frame under high-Mach conditions.
8	NAS08	solves a 2-D Navier-Stokes problem with terms included to account for reactive chemistry.
9	NAS09	solves a supersonic 2-D Navier-Stokes problem.
10	NAS10	solves a 2-D Navier-Stokes problem involving chemistry.
11	MFLOP90	executes 24 common synthetic Fortran kernel loops. The loops range in complexity from easily vectorized, to very difficult. (This code is also known as the Livermore Loops.)
12	NASKERN	7 kernel routines often used in CFD calculations. They include an MxM matrix multiply, a 2-D complex FFT, Choleski factorization, a vectorized block tri-diagonal solver, Gaussian elimination, a vortex generator, and a penta-diagonal matrix solver.

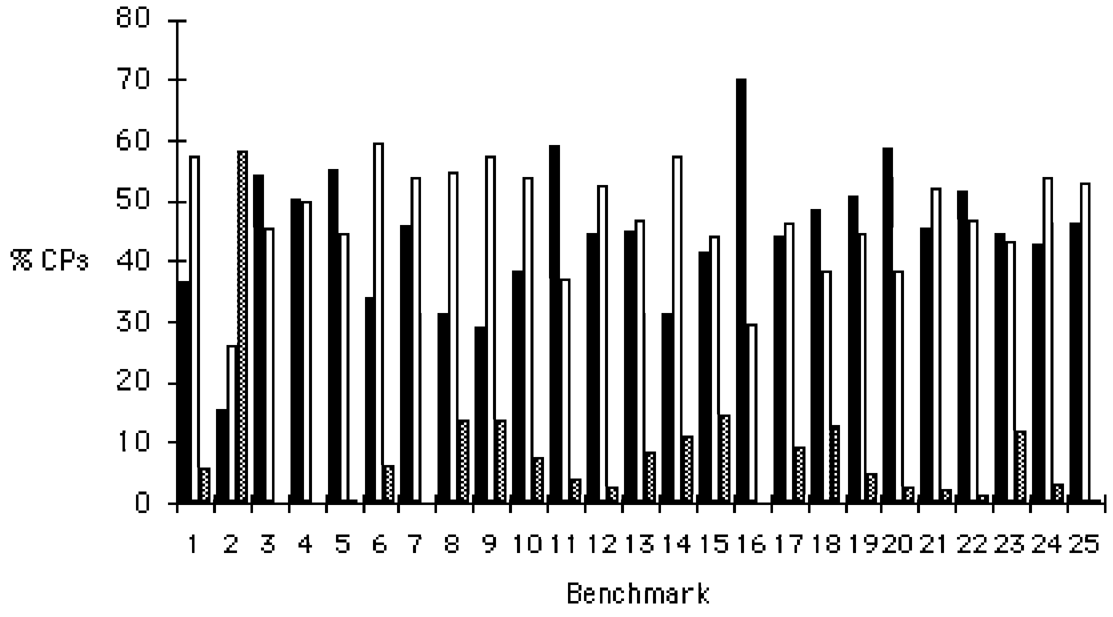
Table 7 — Benchmark Descriptions

13	AMD	a fluid dynamics code used to study air pollution. Its computational kernel relies heavily on FFTs.
14	ARC2D	solves a 2-D supersonic reentry fluid dynamics problem using a sparse linear system solver and a fast elliptic solver.
15	BDNA	a molecular dynamics simulation of a nucleic acid. This code uses an ordinary differential equation (ODE) solver.
16	DYFESM	performs a structural dynamics analysis for an engineering design problem. It uses a sparse linear system solver, a nonlinear algebraic system solver, and an ODE solver.
17	FLO52	solves a 2-D transonic flow fluid dynamics problem. Its computational kernel uses a multigrid scheme with an ODE solver.
18	MDG	performs a molecular dynamics simulation of liquid water, using an ODE solver.
19	MG3D	solves a signal processing problem involving seismic migration. The code uses both FFTs and ODE solvers.
20	OCEAN	2-D fluid dynamics simulation of a section of ocean. This solver uses primarily FFTs.
21	QCD	lattice gauge solution to a quantum chromodynamics problem. A Monte Carlo scheme is used in the solution.
22	SPEC77	another fluid dynamics code—this code performs a weather simulation using FFTs and rapid elliptic problem solvers.
23	SPICE	simulates electronic circuits using sparse linear solvers and ODE solvers.
24	TRACK	another signal processing code—this code performs missile tracking using convolution as the primary mathematical technique.
25	TRFD	uses integral transforms to solve a 2-electron problem from molecular dynamics.

Table 7 — Benchmark Descriptions (cont.)

the percent of clock periods occupied by floating point adds, multiplications, and reciprocals. A full divide requires three multiplies and a reciprocal. Figure 2 clearly shows that these benchmarks are fairly well balanced as far as the additions and multiplies are concerned, but the reciprocal unit is somewhat underutilized.

Figure 2 — FLOP Balance Normalized By CPs



Benchmark I/O requirements were very low, with only a few exceptions. Sixteen benchmarks required less than 1 Mbyte, four used between 1 and 10 Mbytes, and only one (MG3D) used more than 100 Mbytes of I/O. As mentioned earlier, all I/O was mapped to the SSD rather than going out to a rotating disk. Using the SSD reduced the actual time spent in I/O to a minimum.

The benchmarks showed a wide mix of performance when compiled with default vectorization (Figure 3). The codes that have the best performance (over 100 MFLOPS) are all CFD codes that do not use FFTs as solvers (including both 2-D and 3-D codes), kernels from CFD codes (NASKERN), and a molecular dynamics code which employs an ODE solver (BDNA). The FFT based codes, among others, have a relatively moderate level of performance.

Compilation Expense

Code optimizations can only come at the expense of code analysis. Sometimes the analysis is inexpensive, sometimes it is not. The analysis necessary for vectorization is generally a subset of that required for parallelization. In order to provide a basis for comparison, Figure 4 gives the compilation times, in seconds, for each

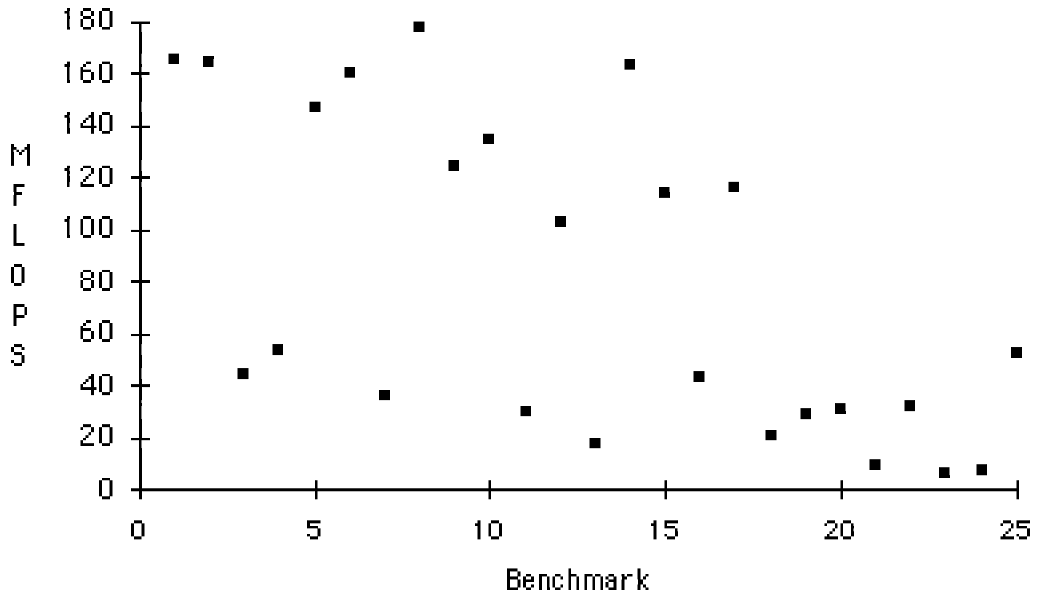


Figure 3 — Default Vectorization Performance

of the benchmarks using only the default CRAY optimizations. All program compilation times are given in Appendix B.

As can be seen from the chart, all codes compiled fairly quickly, even the larger codes like QCD (number 23 in the chart). The question is whether enhanced vectorization or parallelization adds significantly

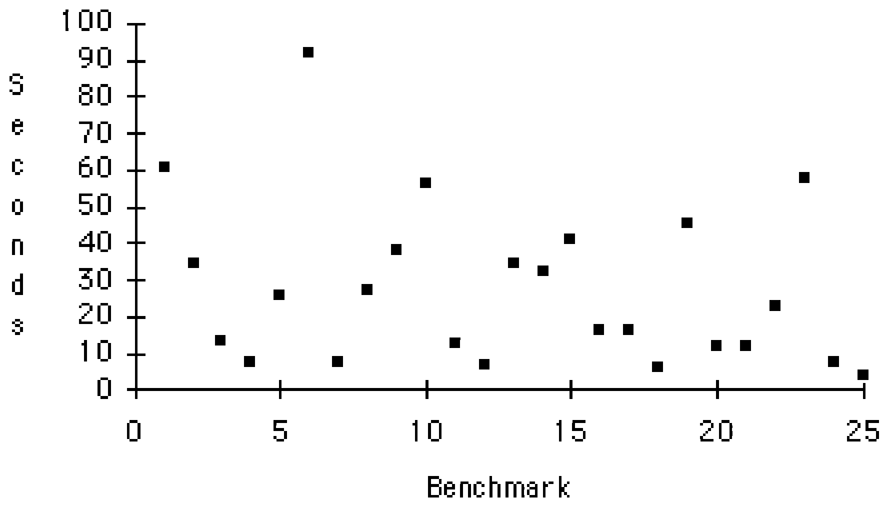


Figure 4 — Compile Times with Default Vectorization

to the compile time. In a research environment like NAS, codes are constantly being modified and recompiled. Thus features that take too long will go unused unless they also significantly shorten the execution time. The following charts show how enhanced vectorization and parallelization increase the compilation time. The shortened execution time will be treated in the next two sections.

The next chart (Figure 5) gives the ratio of enhanced vector compile times (Cv) to default compile times (Cd). From the chart it is clear that fpp adds little to compilation times (about 23%), but KAP/CRAY adds significantly (112% on the average). In one case (NAS09) KAP adds over 350%, boosting the compile time from 38 to 173 seconds, or about three minutes. Even though this is much more time, it is not an intolerable amount of time to wait for a compilation.

Figure 6 shows the expense of compilation using automatic parallelization. The results are similar but more exaggerated than for enhanced vectorization, since the analysis for parallelization is more extensive. For parallelization the average compilation time for fpp is about 3.2 times the default compilation time, while the average KAP time is only a little worse than fpp (3.78). The range for parallel compiles, however, is much larger. Fpp ranges from 48% to 429% more time than default vectorization, while KAP ranges from 88% more to 600% more compile time. The longest KAP compile time is just under 400 seconds (6.5 minutes), which could be tiresome to wait for if it happened too frequently.

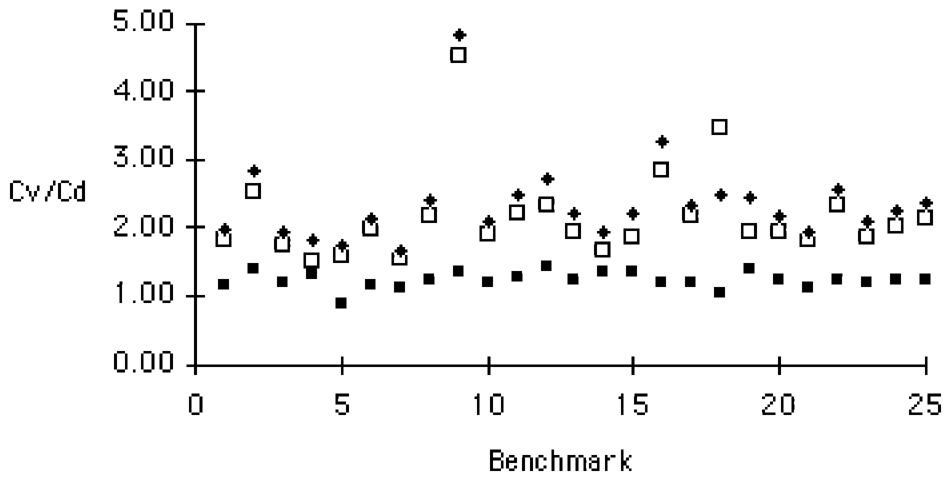


Figure 5 — Enhanced Vector Compilation Time Expense

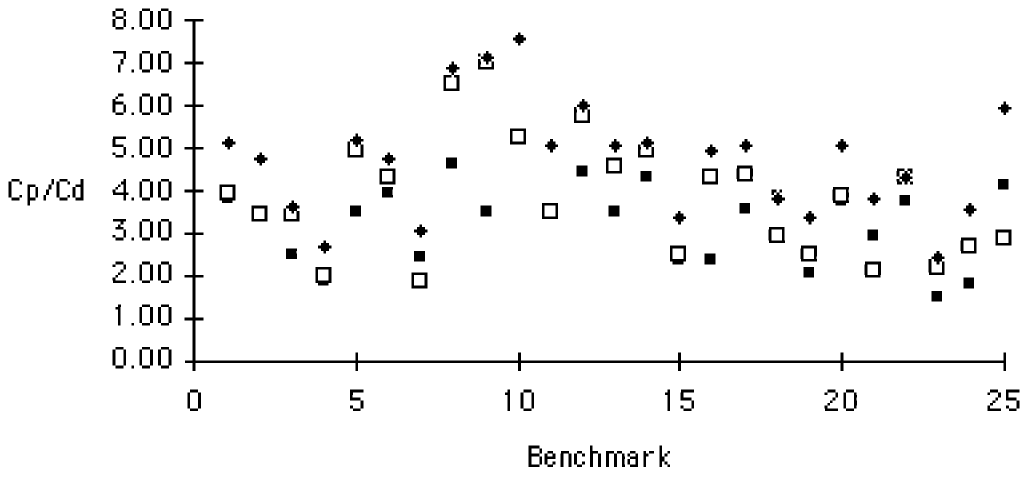


Figure 6 — Parallel Compilation Expense

Enhanced Vectorization

The CRAY Fortran compiler, cft77, vectorizes some types of loop constructs, but in order to keep the compilation time to a minimum it generally uses only those optimizations that are both fast and frequently effective. Fpp, on the other hand, performs a more extensive analysis and therefore has greater potential for improving code performance. Figure 7 shows the performance of all benchmarks, in millions of floating point operations per second (MFLOPS), after enhanced vectorization has been used. Appendix C contains performance information for each of the benchmarks under both default and enhanced vectorizations.

From the chart it appears that there is little change in benchmark performance. Those codes that do well without enhanced vectorization also do well with it, and those that do poorly without are not much improved. Figure 8 summarizes the actual speedups (default vectorization elapsed time divided by enhanced vectorization elapsed time) observed for the individual benchmarks.

Figure 8 seems to indicate that the speedups cluster about 1.00 (i.e., no speedup at all), and indeed this is the case. The average speedup achieved by fpp is 1.02, which is not as good as one might expect. Fpp gained some improvement for 12 of the 25 codes, or just less than half of the benchmarks. We were hoping for at least 25% improvement but none of the codes did even that well, although two

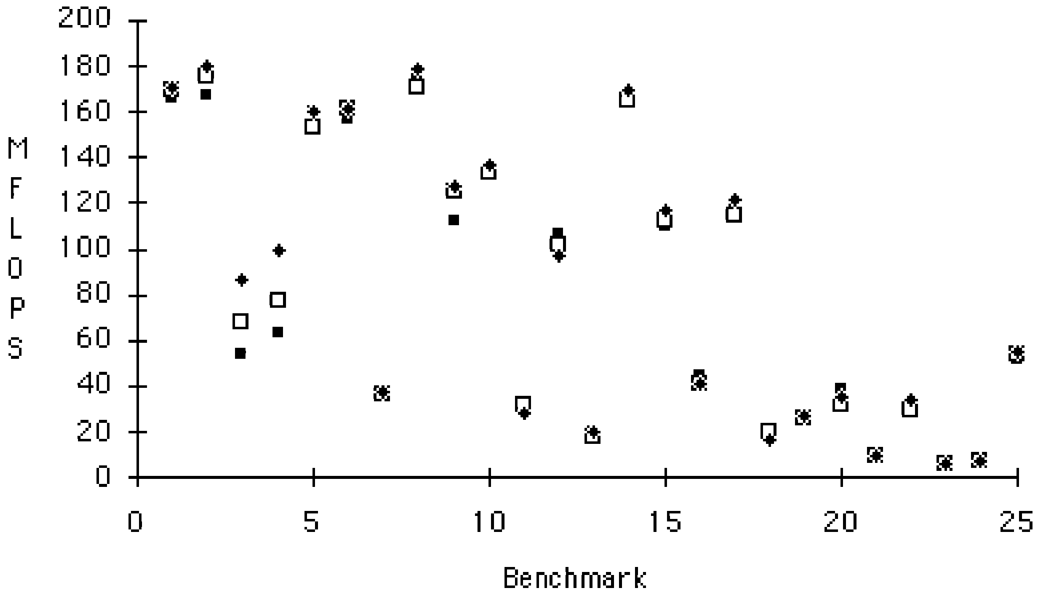


Figure 7 — Enhanced Vector MFLOPS

codes were close. The median speedup was 1.00, the minimum 0.84 (16% slower than no enhanced vectorization), and the best speedup was 1.24. The KAP average was only slightly better but the median was the same as fpp's. Statistics for fpp, KAP, and both are summarized in Table 8.

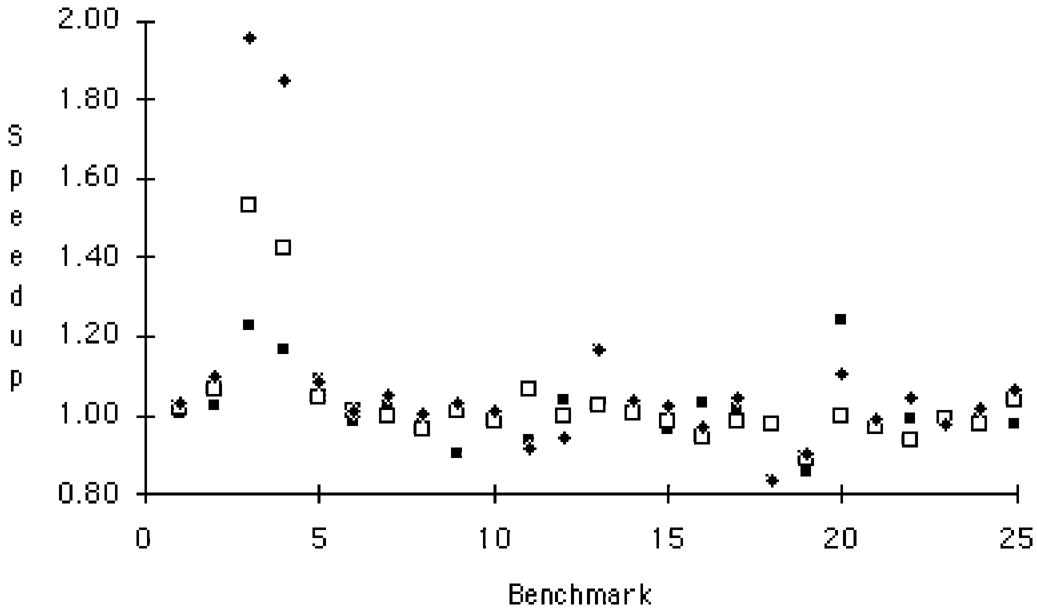


Figure 8 — Speedup from Enhanced Vectorization

	fpp	KAP	Both
Codes Improved	12	10	17
Median Speedup	1.00	1.00	1.03
Average Speedup	1.02	1.03	1.09
Minimum Speedup	0.84	0.89	0.83
Maximum Speedup	1.24	1.53	1.96

Table 8 — Enhanced Vectorization Speedup Statistics

Combining KAP and fpp had some interesting results. The minimum speedup was 0.83—worse than either fpp or KAP alone, but all other statistics showed some improvement. The median increased to 1.03—slightly better than either KAP or fpp alone, while the average speedup was significantly improved. Fully 17 of the 25 codes showed some improvement, and the best speedup was nearly a factor of two over the default vectorization. The following chart (Figure 9) gives the distribution of speedups for fpp only (black column), KAP only (white column), and both (grey column).

The four codes which benefitted most from enhanced vectorization used FFTs as a major part of their computation. Unfortunately, at least one of the FFTs (NAS04) performed well below what it was capable of, even with enhanced vectorization. As a separate experiment, NAS04 was hand optimized by inserting compiler directives above loops which were known to be vectorizable. In every case the apparent dependencies were known to be false — no true dependencies were violated at any time. The elapsed time of

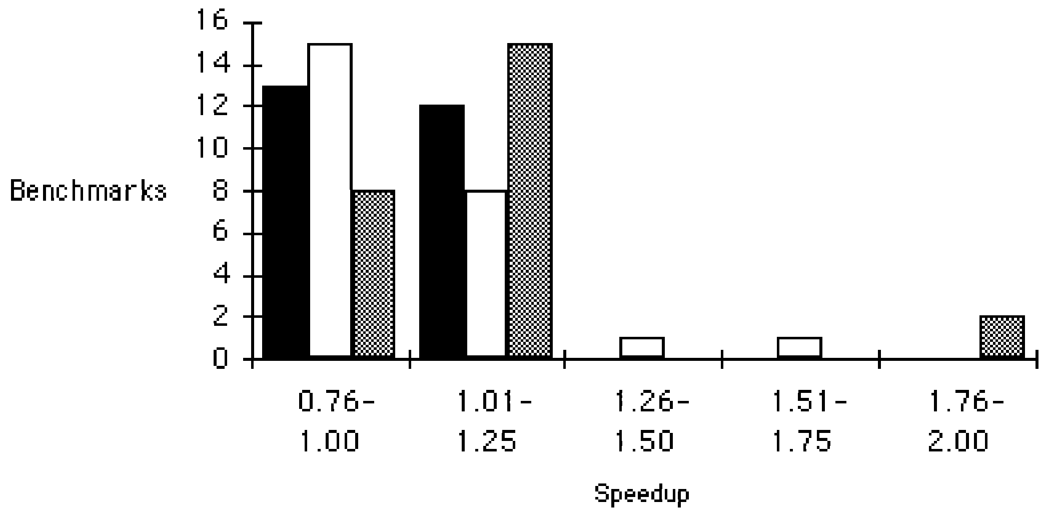


Figure 9 — Enhanced Vectorization Speedup Distribution

the new code was 89.28 seconds, which corresponds to 160.81 MFLOPS, a speedup of 2.89 over the default compiler optimizations, or a factor of 2.55 faster than the original code using enhanced vectorization.

Parallelization

The primary motivation for using parallel processors in scientific work is to increase the computational power available to a user. Unfortunately, one's ability to exploit that power depends on the quantity and granularity of parallelism within the program, and the overhead one must pay to start, suspend, synchronize, and terminate parallel tasks. Figure 10 shows the raw performance of each of the benchmarks using 4 CPUs in its execution. Performance information for the parallel execution of each of the benchmarks is included in Appendix C.

If a program has sufficient parallelism one can expect the speedup to be close to the number of available processors. Figure 11 shows the speedup obtained by using fpp, KAP/CRAY, and both with four CPUs in dedicated time.

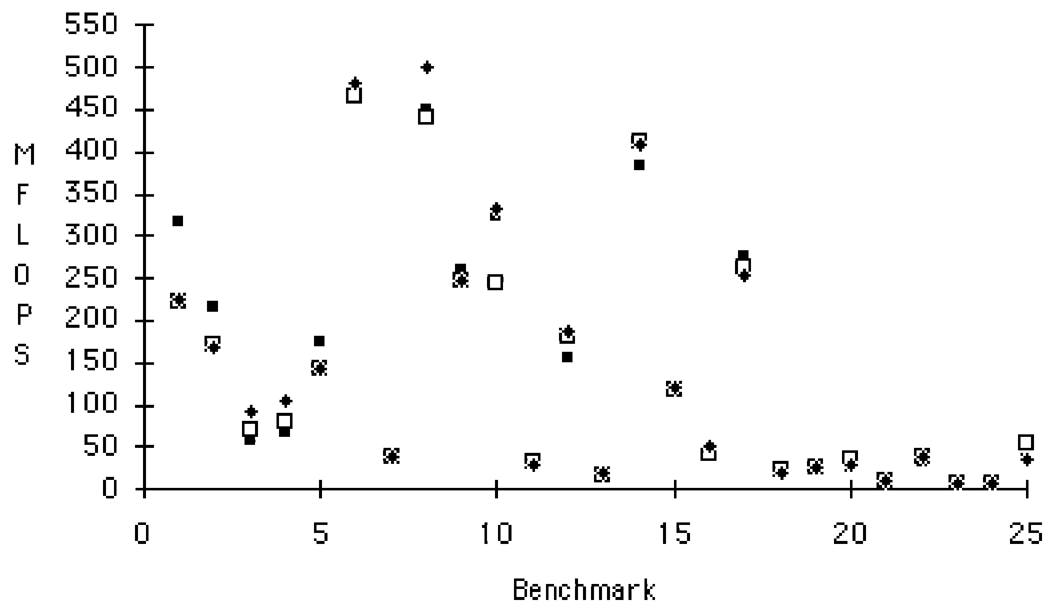


Figure 10 — 4 CPU MFLOPS

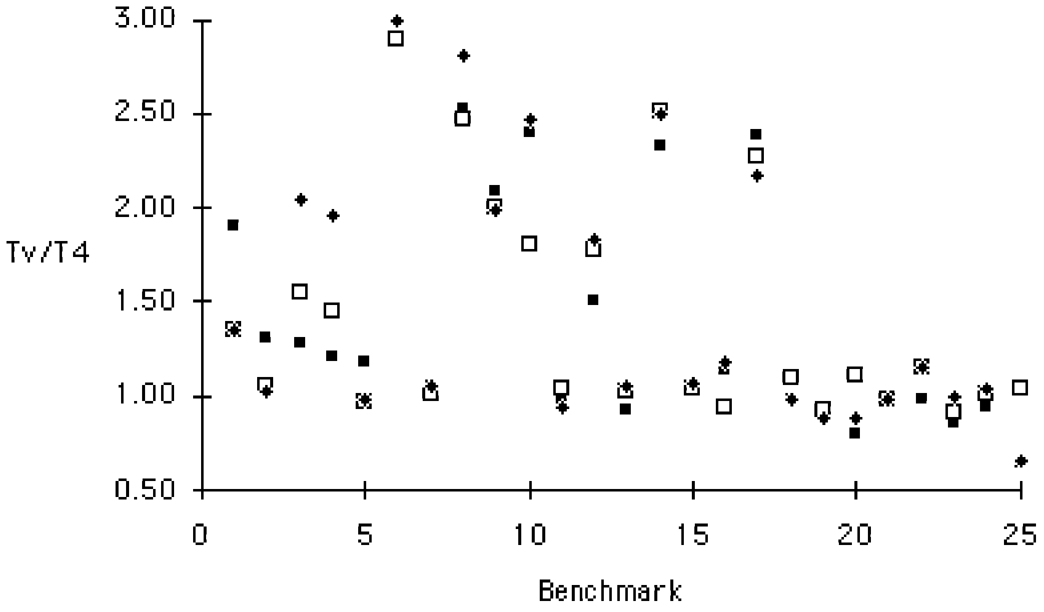


Figure 11 — 4 CPU Speedups

The speedup is calculated using enhanced vectorization as the basis for comparison even though it did not always improve the execution time. Given symbolically, the *n*-processor speedup is defined here as

SS(,n) = F(*TS(,v)*,*TS(,n)*)

where *Tv* and *Tn* are the enhanced vector and *n*-CPU elapsed times, respectively.

This figure shows widely scattered speedups for the benchmarks. It is not too surprising that some codes would parallelize better than others, nor even that the overhead of attempting to parallelize some codes would be greater than the benefits, yielding a speedup less than one. It is valuable to note, however, that many of the codes did show some improvement: 15 benchmarks compiled with fpp showed some improvement over enhanced vectorization, 20 improved with KAP, and 17 improved when both fpp and KAP/CRAY were used. The benchmark performance statistics for the 4 CPU runs are summarized in the Table 9. The distribution of speedups is given in Figure 12.

	fpp	KAP	Both
Codes Improved	15	20	17
Median Speedup	1.14	1.10	1.07
Average Speedup	1.41	1.42	1.48
Minimum Speedup	0.65	0.91	0.66
Maximum Speedup	2.88	2.90	2.99

Table 9 — 4 CPU Speedup Statistics

The FFT codes which improved significantly under enhanced vectorization did not improve as much as might be expected under parallelization. This is in part because even after enhanced vectorization they were relatively poor performers—under 100 MFLOPS on a single CPU. The improvement seen with 4 CPUs matches the improvement gained under enhanced vectorization. The semaphore wait time for these codes is very close to 75%. This indicates that the improvements did not come from parallelization, but rather from the enhanced vectorization which is also present with the parallelization.

The codes which performed well under parallelization were 2-D CFD codes and one 3-D CFD code that did *not* use FFTs in their computational kernel. Each of these codes were in the 100 to 200 MFLOPS range, but the fact that these codes are 2-D CFD codes is more important than their initial high performance. Other codes displayed equally high performance under enhanced vectorization but did not perform as well under parallelization.

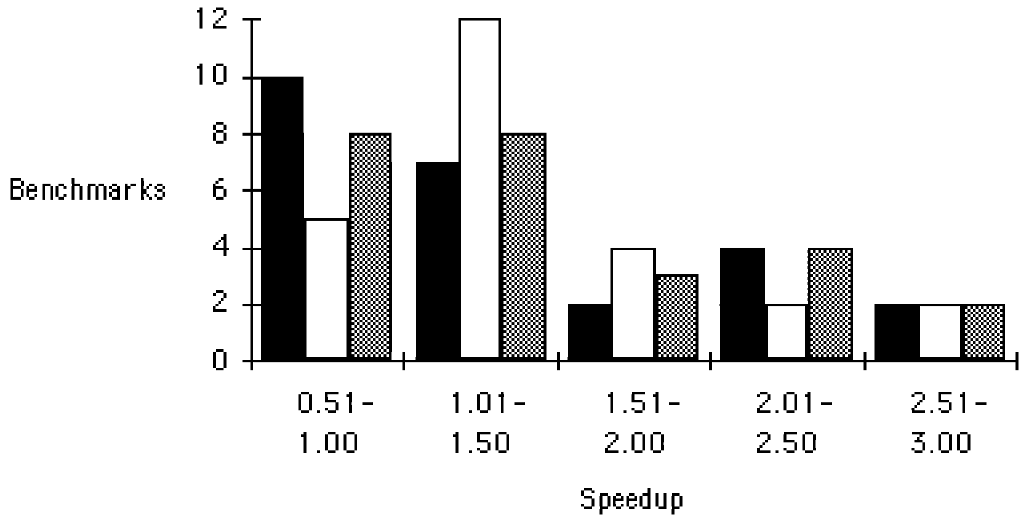


Figure 12 — 4 CPU Speedup Distribution

A separate experiment was conducted in which NAS04 was hand optimized, as it was for enhanced vectorization. The result was that its elapsed time dropped to 51.84 seconds, with 101.43 seconds of CPU time on 4 CPUs. This corresponds to 276.93 MFLOPS, or more than 4 times better than any execution without hand optimization.

If one compares the 4-CPU distribution against the enhanced vectorization speedup distribution, it is readily apparent that they are similar, in that the majority of the codes show less than a 50% improvement. They also differ in that a greater number of benchmarks show an improvement greater than 50% (i.e., a speedup greater than 1.50). Another difference is that the codes which make up the high end of vectorized codes are FFTs, where the CFD codes were improved by parallelization and FFTs were not.

Each of the benchmarks was also run with 8 CPUs to see how the performance would differ from 4 CPUs. The raw performance (in MFLOPS) is given in Figure 13. Figure 14 shows the speedup gained by using 8 CPUs. The statistics are summarized in Table 10, and the distribution of speedups is given in Figure 15.

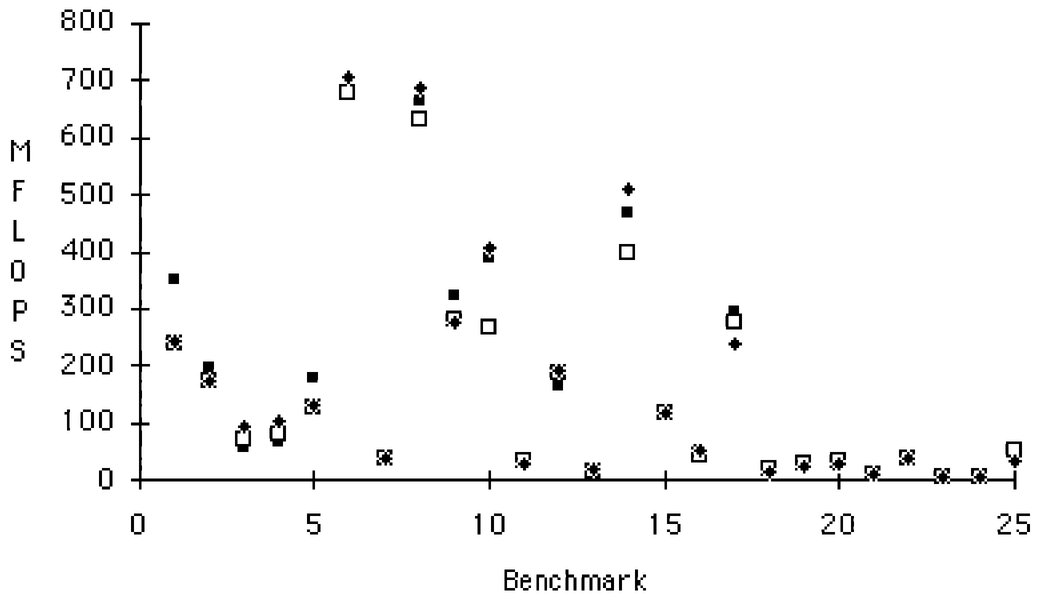


Figure 13 — 8 CPU MFLOPS

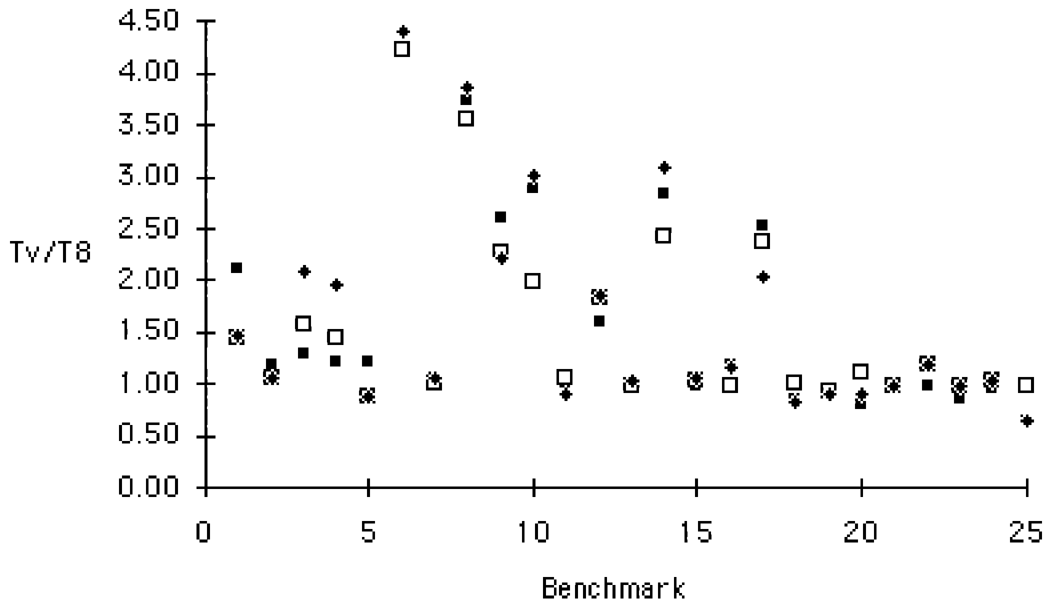


Figure 14 — 8 CPU Speedups

The 8 CPU speedup scatter plot (Figure 14) shows several things of interest. It shows that the codes which improved the most with 8 CPUs were the same codes that improved the most with 4 CPUs, namely, CFD codes. It is also clear that the efficiency is dropping rapidly with an increasing number of CPUs. Parallel efficiency is defined as

$$ES(,n) \quad = \quad F(SS(,n),n) \quad = \quad F(TS(,v),n\ TS(,n))$$

(For example, the 8-CPU efficiency is *Tv*/(8**T*8)). The best efficiencies obtained by the 4 CPU runs were close to 75%, the equivalent of using 3 out of 4 CPUs. The highest 8 CPU efficiencies were lower—about 50%, or the equivalent of using 4 CPUs out of 8. Essentially, we had to double the number of CPUs in order to buy that additional 25% improvement.

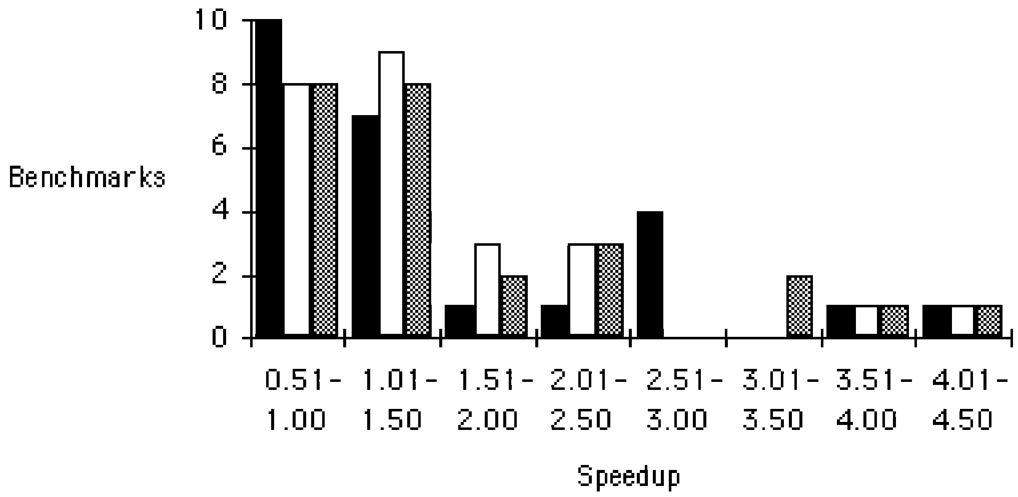


Figure 15 — 8 CPU Speedup Distribution

Parallelism and Overhead

Fpp and KAP/CRAY exploit parallelism found in program loops. When nested loops are encountered it is generally more effective to vectorize the inner loops and distribute outer loop iterations across the available processors. Vectorization should never be sacrificed for the sake of large grain parallelism, because of the low initial overhead and the high potential gain vectorization offers. Data dependencies may interfere with both the vectorization and parallelization of loops. If the loop is large and highly vectorizable, it is often parallelizable.

Given the speedup and the number of processors, we can calculate the effective parallel fraction of a code. Amdahl's Law says that the *n*-processor speedup (*Sn*) is related to the fraction of the operations which can be performed in parallel (*p*) and the number of processors (*n*) by the equation

$$SS(,n) = F(1,1-p+F(p,n))$$

Amdahl's Law assumes that the parallel operations can be exploited without incurring any penalty, and the operations are continuous rather than discrete. In MIMD systems neither assumption is true,

	fpp	KAP	Both
Codes Improved	15	17	17
Median Speedup	1.18	1.06	1.06
Average Speedup	1.59	1.53	1.63
Minimum Speedup	0.64	0.87	0.66
Maximum Speedup	4.24	4.23	4.40

Table 10 — 8 CPU Speedup Statistics

but the assumption of no overhead is most important. The microtasking library incurs overhead in creating, suspending, restarting, synchronizing, and terminating parallel tasks. The overhead incurred within a code will depend on the number of processors used, and the number, size, and structure of each loop. Thus Amdahl's Law would have to have a term to account for parallel overhead which varied with the conditions of execution. However, if the overhead varies only slightly with the number of processors, one can at least obtain the *effective parallel fraction*, which will give some indication of a code's parallel performance, by solving Amdahl's Law for *p*.

$$p = \frac{n}{n-1}\left(1 - \frac{1}{S_n}\right) = \frac{n}{n-1}\left(1 - \frac{T_n}{T_v}\right)$$

The next chart (Figure 16) shows the effective parallel fraction for each of the benchmarks using 4 CPUs. When we report values of the parallel fraction *p* for a given program, we will generally present it as a percentage, e.g., *p*=0.43 would be reported as 43%. Notice also that the parallelism is negative if the application is slowed by parallelization. This can occur when the overhead of initiating and terminating a parallel task is larger than the benefit gained.

The overhead for *n* processors is defined here as

$$\text{Overhead} = 100\,\frac{(User_n+Sys_n)-(User_v+Sys_v)}{(User_v+Sys_v)}$$

where *User* and *Sys* are the user and system times, respectively, for the *n*-CPU parallel runs (*n*) or the enhanced vector runs (*v*). Overhead for the 4 CPU runs is shown in Figure 17.

Loops that vectorize well do so because the iterations, or at least parts of the iterations, can be executed independently. If there is enough independent work in these loops, the iterations can be divided between multiple processors for a net gain in speed. How much work is "enough" is determined by the overhead involved in starting up the parallel processors and dividing the loop iterations. This overhead is in addition to what a serial version of the same program would do, so it necessarily increases the work load of the machine.

Very long loops permit the overhead to be amortized over many iterations, which improves the overall performance of the code. The lengths of the dominant loops were measured for several codes, and the results are recorded in Table 11. Loop lengths are measured in iterations. (Recall that NAS04 is an FFT code, while the others are CFD codes.) The parallel fraction for NAS04 is from the automatically parallelized version. The hand optimized version has a parallel fraction of 55%.

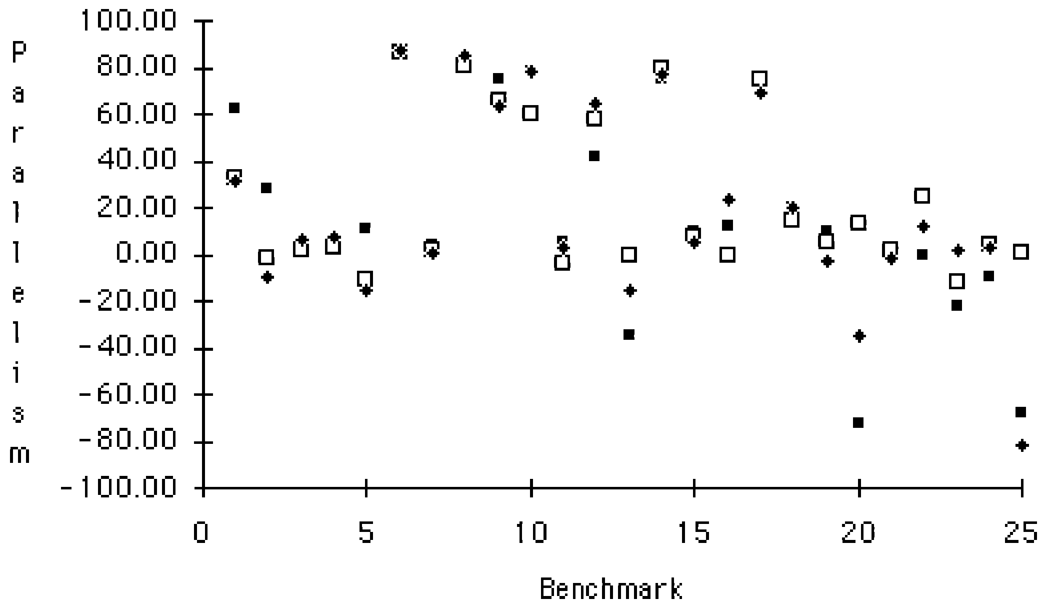


Figure 16 — % Parallelism in 4 CPU Runs

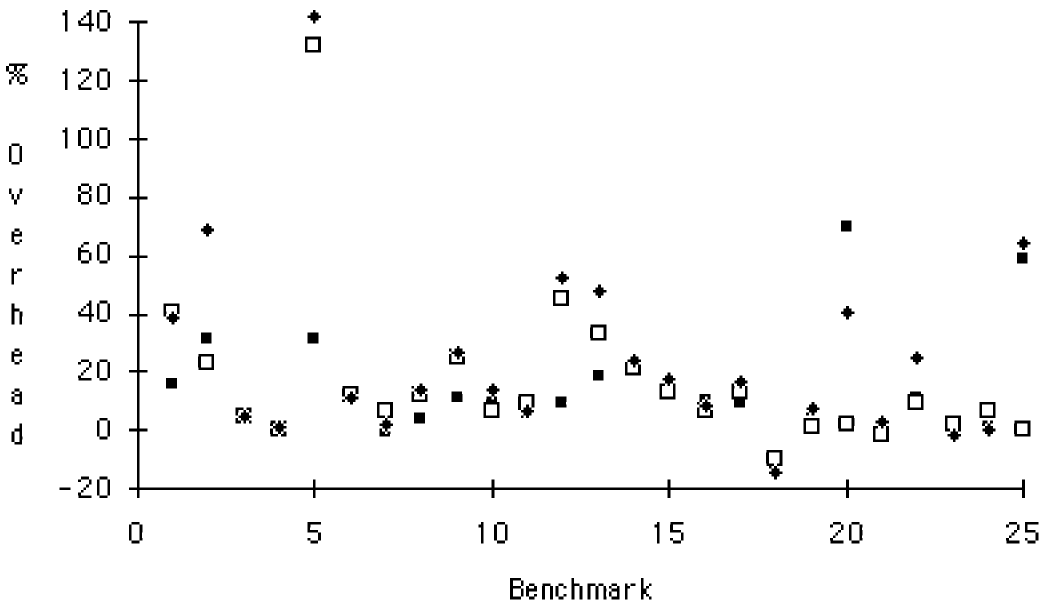


Figure 17 — % Overhead in 4 CPU Runs

The overhead is especially important to the throughput of a machine. A 100% overhead means that as much time is being spent maintaining parallelism as is accomplishing useful work. It also means the throughput could be reduced by 50%. The overhead is very high for many codes, most of which do not parallelize well. Not surprisingly, the codes which do parallelize well have low levels of overhead. If only the high performance codes were used (those with speedups greater than 2.00), the average overhead would be around 20%. This would reduce the potential throughput by about 17% over serial execution. This might be acceptable if there are enough idle CPU cycles, in which case the result could be a shorter average turnaround time for jobs in the machine.

Code Name	Loop Length	Parallel Fraction
NAS04	60	5 %
NAS01	100	63 %
NASKERN	256	42 %
NAS06	1,000	88 %
NAS08	11,000	81 %
NAS10	26,500	79 %

Table 11 — Loop Lengths and Parallel Fractions

As mentioned before, the overhead does vary with the number of CPUs used. Some of the poorly parallelizing codes suffer substantial increases in overhead when going to 8 CPUs, but that doesn't matter much since such codes would probably not be used in their parallel form. The highly parallel codes, however, add approximately 15% overhead by going from 4 CPUs to 8. The effective parallelism and overhead for the 8 CPU runs are shown in Figures 18 and 19.

Percent Vectorization

While it is possible, in theory, to parallelize some poorly vectorized code, no evidence was found in this study that would indicate such was taking place. This may be because such opportunities might not exist within the selected benchmarks, or because the tools themselves might not be set up to do that. While high vectorization levels seem to be required in order to parallelize a program, it is not a guarantee of success. Several codes, e.g. NAS05 and NASKERN, had higher levels of vectorization than their levels of parallelization would suggest. (Percent vectorization for each benchmark is calculated in Appendix D.)

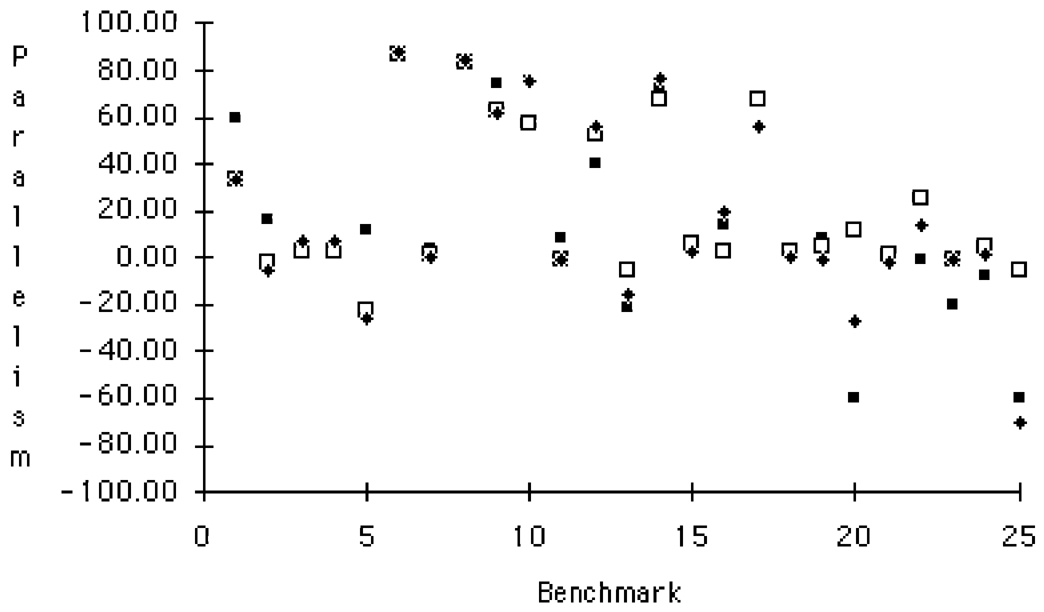


Figure 18 — Parallelism in 8 CPU Runs

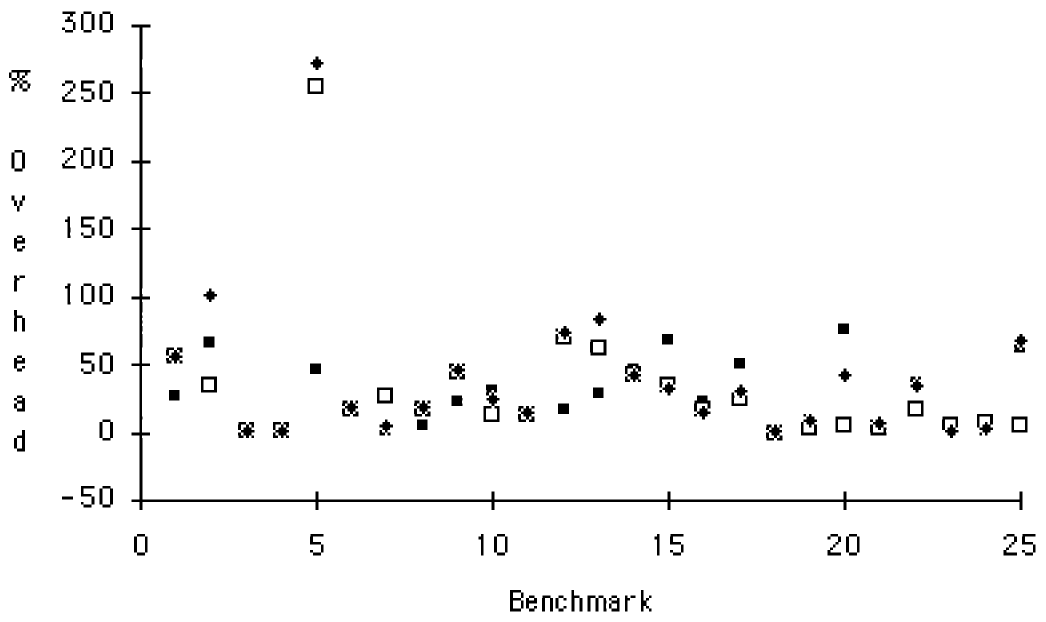


Figure 19 — % Overhead in 8 CPU Runs

Figure 20 shows the relationship between percent vectorization and percent parallelization for the 4 CPU runs. This chart shows that the effective parallelism is clearly related to the percent vectorization, although the correlation is not direct. Percent vectorization appears to be a least upper bound of the percent parallelization, thus it can be substituted directly into Amdahl's Law to obtain a tight upper bound on parallel speedup. The parallelism values for the 8 CPU runs are very nearly the same as for the 4 CPU runs, so no separate chart is presented. Percent parallelism for each benchmark is given in Appendix E.

Each of the remaining charts (Figures 21 through 25) emphasize the high level of vectorization needed for a significant performance. At the same time they remind us that high vectorization is no guarantee
of good parallel performance. It is a necessary but not a sufficient condition for parallelization. Figures 21 and 22 show the parallel speedup as a function of vectorization. Figures 23, 24, and 25 show the performance in MFLOPS as a function of vectorization. The steep curve associated with Amdahl's Law can be seen in 24 and 25.

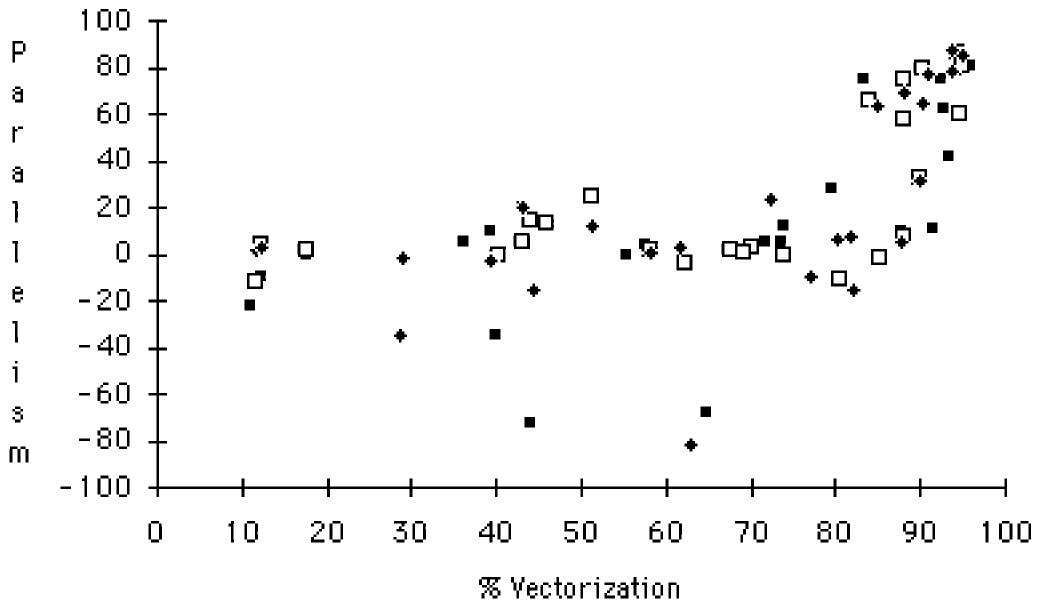


Figure 20 — 4 CPU % Parallelism vs. % Vectorization

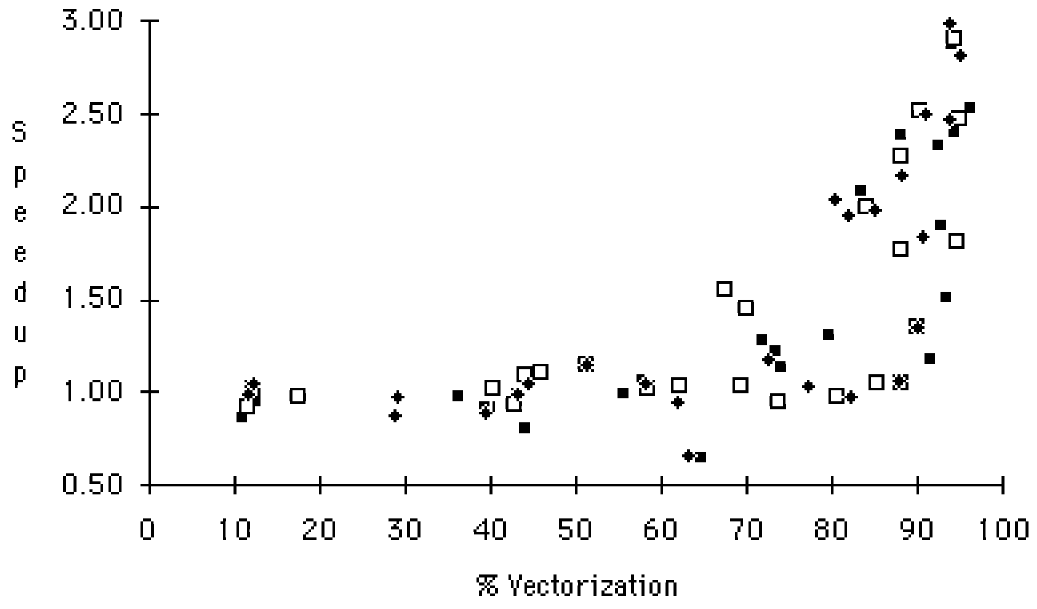


Figure 21 — 4 CPU Speedup vs. % Vectorization

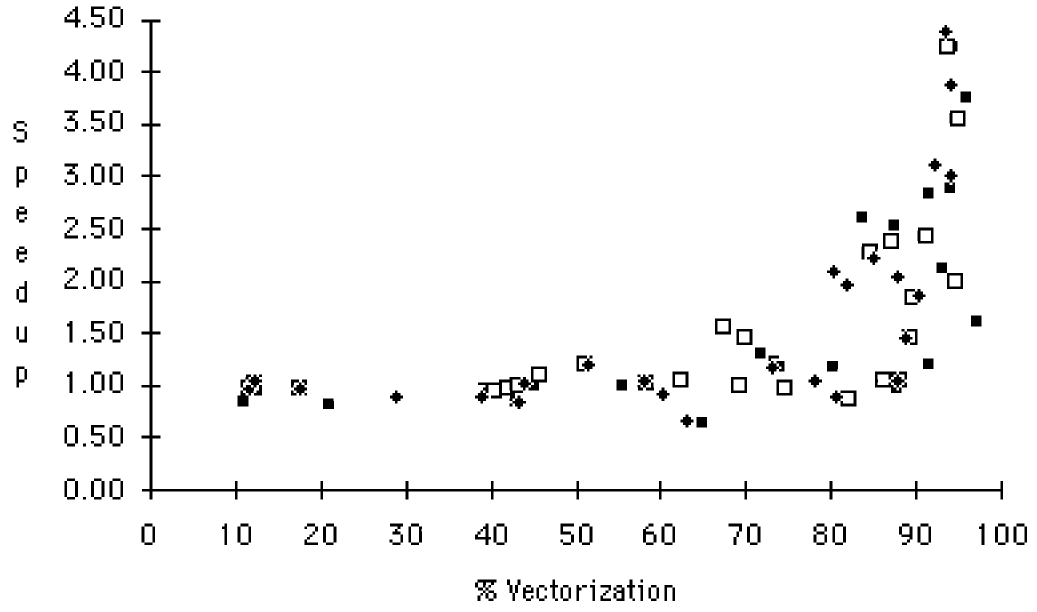


Figure 22 — 8 CPU Speedup vs. % Vectorization

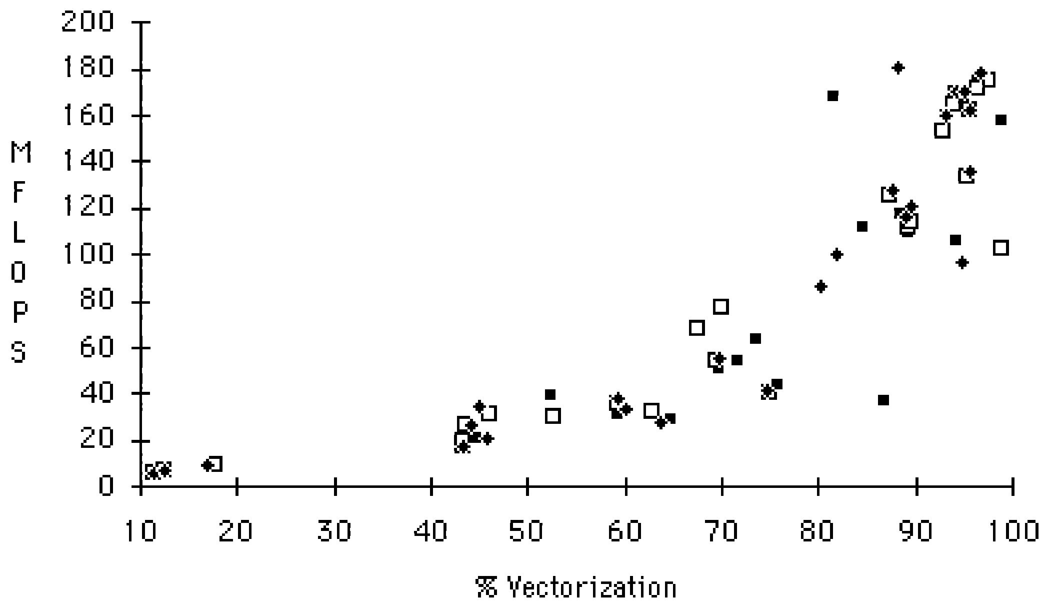


Figure 23 — Enhanced Vectorization

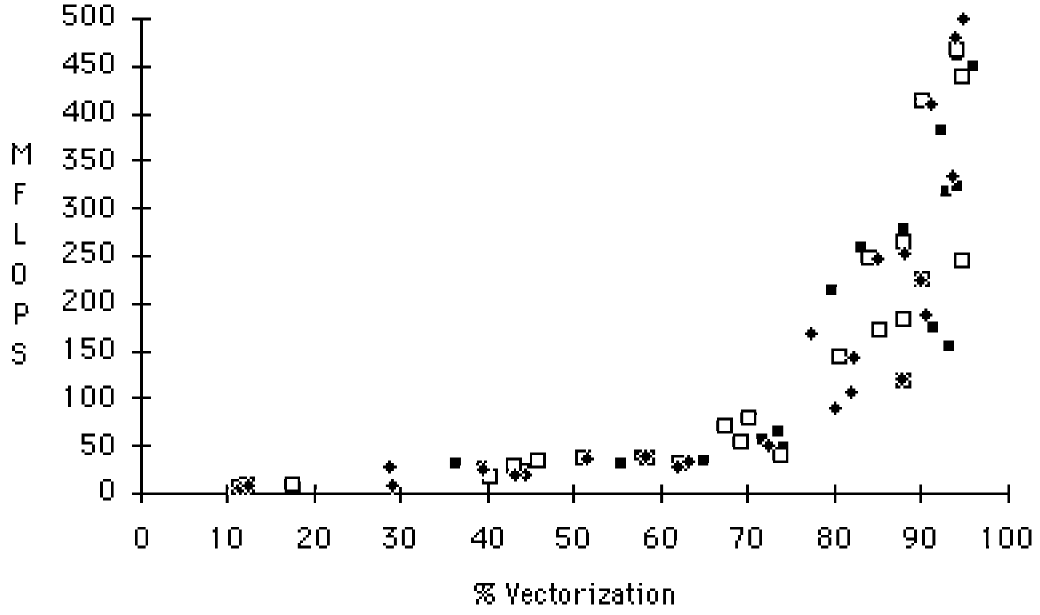


Figure 24 — 4 CPU MFLOPS

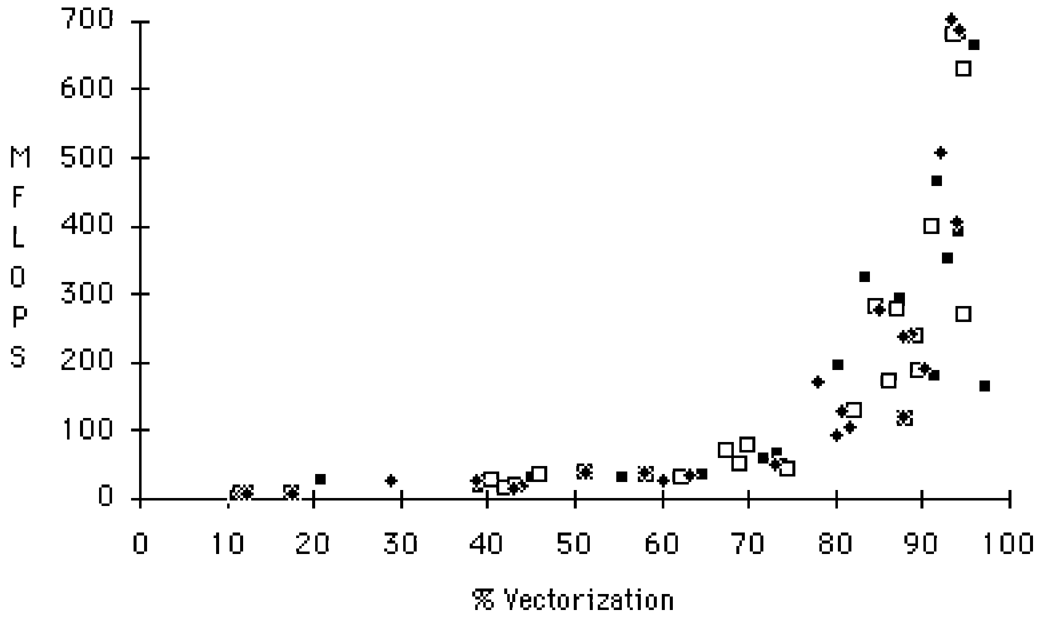


Figure 25 — 8 CPU MFLOPS

Compilation Problems

Neither fpp nor KAP were completely free of problems. KAP had more problems than did fpp, probably because the writers of fpp had greater access to the system, particularly fmp, the program which translates microtasking directives into CRAY library calls. All of the bugs we discovered in both products were annoyances, but could be tolerated using a simple work around. None were so serious as to render the product unusable, and all seemed to be fixable with no more than a moderate amount of effort. Below we give a brief summary of the bugs we found.

- KAP incorrectly declares CDIR$ SHORTLOOP on some variable length loops, which may give either incorrect results or a floating point exception.

- KAP leaves DATA statements within CMIC$ PARALLEL sections, which causes CRAY Autotasking to fail during compilation. One might argue that DATA statements may not legally occur where CMIC directives are allowed, but neither does KAP flag it as an error. Either moving DATA statements to the beginning of a routine or flagging them as being out of place would be acceptable.

- KAP occasionally uses KAP-generated indexing functions or arrays, but doesn't provide a definition for them. This causes undefined external references to occur in the program.

- KAP does not always provide SHARED or PRIVATE declarations for array index variables it introduces into the program. Parallel programs are very sensitive to whether certain variables are private or shared, and fmp is not able to decide which mode is correct.

- Fpp does not always distinguish between comments that are compiler directives and those that are not. As a result, fpp sometimes rearranges microtasking directives which should be left in place.

Conclusions

Using fpp to enhance the vectorization does not significantly slow down the compilation process. It added, on the average, only 23% to the compilation times of our benchmark suite. On the other hand, neither does it significantly improve its performance. The FFT codes improved the most, but the best improvement was only 24%.

In contrast, using KAP for enhanced vectorization does slow program compilation down considerably, adding on the average 112%, and sometimes as much as 350% to the compilation time. To its credit, it does speed up FFTs significantly. In two cases, FFT benchmarks were improved by approximately 50%. Combining fpp with KAP brought about an even greater improvement—almost 100%. Like fpp, though, non-FFT codes did not noticeably improve.

Both KAP and fpp substantially slowed the compilation time when they were used for automatic parallelization, and KAP was usually only a little slower than fpp. Both improved the performance of CFD codes, especially the 2-D CFD codes. Used together, the improvement was a factor of 2.5 to 3.0 using 4 CPUs, and from 2.5 to 4.5 on 8 CPUs. Neither fpp nor KAP had a significant performance edge over the other.

Neither KAP nor fpp were able to uncover sufficient parallelism in CFD codes for highly parallel execution. Parallelism ran about 60% to 90% which would correspond to maximum speedups of 2.5 to 10, given 1) an infinite supply of parallel hardware, and 2) no increase in the overhead to use it. As good as this is, we anticipate far greater levels of parallelism (95% to 99.9%) will be needed to take advantage of future parallel hardware. The overhead required to maintain parallel execution was relatively large, and reducing this might significantly improve these values.

Parallel efficiency was much higher for 4 CPU runs (75%) than for 8 CPU runs (50%), primarily due to Amdahl's Law and the modest amount of parallelism found in the programs. It is a corollary to Amdahl's Law that the efficiency will always be higher for the 4 CPU runs than the 8 CPU runs. It is because of the moderate levels of parallelism that the parallel efficiency differs so greatly.

Overhead from parallelization was high, to the extent that widespread parallelization could significantly reduce a system's throughput in a production environment. Overhead was especially high for codes that did not parallelize well. Overhead for the CFD codes on 4 CPUs averaged approximately 15% to 20%. On 8 CPUs the average increased to about 25%.

Program vectorization and Amdahl's Law determine an upper bound on speed improvements due to parallelization. Parallelized codes are highly vectorized codes, but a high level of vectorization does not guarantee that a code will also parallelize. The need for high levels of vectorization grows dramatically as the number of CPUs increases.

Because the levels of parallelism KAP and fpp found in even the best codes were much lower than desired, we conclude that such tools will probably not be able to find sufficient parallelism without human assistance. In our opinion the ideal tool for parallelizing Fortran programs would have:

- Extensive dependence analysis — this has been a traditional area of focus, and allows the tool to distinguish, albeit imperfectly, between code which is parallelizable and code that must remain sequential. Both KAP and fpp seem to do quite well at this. The most obvious problem was that in order to correctly determine that certain dependences were false, both tools would have needed information about the program's runtime behavior, or about its algorithms. More extensive analysis might help some, but it can't use information that isn't there without a programmer's help.

- Extensive code transformation vocabulary — another area of traditional focus, it gives the tool the ability to improve the level of parallelism or the granularity of parallelism while retaining the original meaning of the program. Again, KAP and fpp seem to do well at this.

- Runtime statistics — allows the tool to steer the user towards the most heavily used, and therefore most profitably optimized sections of code. It also could retain program traces and branch probabilities, which might further assist in choosing appropriate code restructuring optimizations.

- Queriable interface — allows the tool to display only the information that the user wants to see, instead of all information that *might* be relevant. It would be beneficial if the interface were sufficiently robust to be able to identify user supplied patterns in the code, array indices, and variable usages.

- Analysis facility — indicates the circumstances under which perceived dependences are *false* so the programmer can decide whether that section of code is safely parallelizable.

- Suggestion facility — makes optimization and parallelization suggestions to the user, and indicates the circumstances under which the suggestions are valid. This should include an understanding of the vectorization and parallelization overhead involved, letting the user know when the section of code being parallelized may not have sufficient work to overcome the expected cost.

Acknowledgements

We would like to thank all those who helped out with the preparation of this report, and as is always the case, there are several who merit special recognition. Robert Bergeron and Russell Carter contributed significantly to the ideas in this report, and steered us away from numerous pitfalls in the data gathering. Kuck and Associates generously allowed the use of their software and provided the necessary support, which we appreciate greatly. Last but not least, Eugene Miya contributed to the philosophical foundations of this work.

References

[1] L. Kipp, "Perfect Benchmarks Documentation, Suite 1," Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Urbana, IL, 1990.

[2] "CRAY Y-MP Computer Systems Functional Description Manual," HR-4001, CRAY Research, Inc., Mendota Heights, MN, 1988.

[3] "CF77 Compiling System, Volume 4: Parallel Processing Guide," SG-3074 4.0, CRAY Research, Inc., Mendota Heights, MN, 1990.

[4] "KAP/CRAY User's Guide," Kuck & Associates, Inc., Champaign, IL, 1989.

Appendix A — Benchmark Characteristics

This appendix lists several characteristics for each of the benchmarks. They are:

Source lines — number of text lines in the original program, including comments and blank lines. No compiler directives are included.

Size — total program memory requirements, in megawords (2**20 64-bit words), as obtained by compiling the program for sequential execution by the CRAY Fortran compiler, cft77. Memory size was measured by the UNICOS utility "size".

Floating point adds, multiplies, and reciprocals — total operations in each category as measured by the Y-MP hardware performance module (group 0).

Data transferred — total program I/O requirements including all raw and formatted read and write operations. The units are megabytes.

Appendix B — Compilation Data

This appendix contains the CPU times used to compile each of the programs, both for parallel and sequential execution. User and system times were recorded from the Unix C-shell "time" facility.

Appendix C — Performance Data

This appendix contains the following information:

Elapsed sec — total program elapsed (wall clock) execution time, in seconds, as measured by the UNICOS "ja" utility.

CPU sec — program CPU (user) execution time, in seconds, as measured by the "ja" utility.

System sec — program system time, in seconds, also from "ja".

% Vectorization — percentage of program instructions that were vector instructions. This value is calculated in Appendix D.

MFLOPS — millions (10**6) of floating point operations per second. It is calculated by dividing the total floating point operations (recorded in Appendix A) by the elapsed time.

Sem. Wait — semaphore wait time as reported by the hardware performance monitor (HPM) group 1. Semaphore wait time indicates the percentage of time CPUs spend in a "parked" state waiting for work to become available.

Con. CPU — average concurrent CPUs as reported by the "ja" facility. It indicates the average number of CPUs that are either busy in behalf of the program or idle available to do work.

Speedup — multiplication factor of improvement in elapsed time over the default vectorization execution elapsed time. Values below 1.00 indicate the performance was higher with default vectorization.

Instructions — total vector and scalar instructions executed, as recorded by the HPM (group 0).

MIPS — millions (10**6) of instructions executed per second.

CPI — average number of clock periods per instruction, or

CPI = F(Elapsed time,Instructions x 6.0 ns/clock)

Appendix D — Vectorization Data

This appendix contains the calculation of the percent vectorization reported in Appendix C. Block memory reference counts come from the hardware performance monitor (HPM) group 2. All other values come from HPM group 3. Group 3 measurements were obtained with avl disabled.

Block memory (a) — block memory transfers (vector loads and stores).

Vector I+L (b) — integer and logical vector instructions.

Vector Float (c) — vector floating point instructions (additions, multiplications and reciprocals).

Jump/Special (d) — branch, conditional, and special instructions.

Scalar FU (e) — scalar functional unit instructions.

Scalar memory (f) — scalar register load and store instructions.

% Vectorization — percent of program instructions that were vector instructions, calculated by

$$\% \text{ Vectorization} = 100 \ F(a+b+c, a+b+c+d+e+f)$$

Avg Len-I+L — average vector length of all integer/logical vector operations.

Avg Len-Float — average vector length of all floating point vector operations.

Appendix E — Effective Parallel Fraction

This appendix gives the effective parallel fraction for each of the parallel program executions. The formula to compute the parallel fraction is obtained by solving Amdahl's Law, giving

% *p* = 100 F(n,(*n*-1))*B(*1 - F(TS(,n),TS(,v)))

where *Tn* and *Tv* are the elapsed times of the *n*-CPU and vector executions, respectively.